Exp No: 4

TIC TAC TOE Using Minimax And Alpha Beta Pruning Algorithm.

Srinidhi.V
18C102

Aim:

To show TICTACTOE game using minimax & alpha beta pruning algorithm.

Algorithm:- Minimax

* An evaluation function is defined in which its possible to search the whole tree till the leaves.
* It has 3 possible values = (-1) if player that seeks min wins, (0) if its a tie & (1) if the player that seeks max wins.
* Function to check if move is legal is defined and also check if game is over.
* The AI playing seeks two things - to maximize its own score and to minimize ours. max() method makes optimal decision. min() serves as a helper for us to ~~maintain~~ minimize the AI's score.
* Game loop is defined to play TIC TAC TOE game.

Program:

Minimax:

```
import time

class Game:
  def __init__(self):
    self.initialize_game()

  def initialize_game(self):
    self.current_state = [['.','.','.'],
                          ['.','.','.'],
                          ['.','.','.']]

    # Player X always plays first
    self.player_turn = 'X'

  def draw_board(self):
    for i in range(0, 3):
      for j in range(0, 3):
        print('{}|'.format(self.current_state[i][j]), end=" ")
      print()
    print()

# Determines if the made move is a legal move
def is_valid(self, px, py):
  if px < 0 or px > 2 or py < 0 or py > 2:
    return False
  elif self.current_state[px][py] != '.':
    return False
  else:
    return True

# Checks if the game has ended and returns the winner in each case
def is_end(self):
  # Vertical win
  for i in range(0, 3):
    if (self.current_state[0][i] != '.' and
      self.current_state[0][i] == self.current_state[1][i] and
      self.current_state[1][i] == self.current_state[2][i]):
      return self.current_state[0][i]
```

```
    # Horizontal win
    for i in range(0, 3):
        if (self.current_state[i] == ['X', 'X', 'X']):
            return 'X'
        elif (self.current_state[i] == ['O', 'O', 'O']):
            return 'O'

    # Main diagonal win
    if (self.current_state[0][0] != '.' and
        self.current_state[0][0] == self.current_state[1][1] and
        self.current_state[0][0] == self.current_state[2][2]):
        return self.current_state[0][0]

    # Second diagonal win
    if (self.current_state[0][2] != '.' and
        self.current_state[0][2] == self.current_state[1][1] and
        self.current_state[0][2] == self.current_state[2][0]):
        return self.current_state[0][2]

    # Is whole board full?
    for i in range(0, 3):
        for j in range(0, 3):
            # There's an empty field, we continue the game
            if (self.current_state[i][j] == '.'):
                return None

    # It's a tie!
    return '.'

# Player 'O' is max, in this case AI
def max(self):

    # Possible values for maxv are:
    # -1 - loss
    # 0  - a tie
    # 1  - win

    # We're initially setting it to -2 as worse than the worst case:
    maxv = -2

    px = None
    py = None

    result = self.is_end()
```

```
    # If the game came to an end, the function needs to return
    # the evaluation function of the end. That can be:
    # -1 - loss
    # 0  - a tie
    # 1  - win
    if result == 'X':
        return (-1, 0, 0)
    elif result == 'O':
        return (1, 0, 0)
    elif result == '.':
        return (0, 0, 0)

    for i in range(0, 3):
        for j in range(0, 3):
            if self.current_state[i][j] == '.':
                # On the empty field player 'O' makes a move and calls Min
                # That's one branch of the game tree.
                self.current_state[i][j] = 'O'
                (m, min_i, min_j) = self.min()
                # Fixing the maxv value if needed
                if m > maxv:
                    maxv = m
                    px = i
                    py = j
                # Setting back the field to empty
                self.current_state[i][j] = '.'
    return (maxv, px, py)

# Player 'X' is min, in this case human
def min(self):

    # Possible values for minv are:
    # -1 - win
    # 0  - a tie
    # 1  - loss

    # We're initially setting it to 2 as worse than the worst case:
    minv = 2

    qx = None
    qy = None

    result = self.is_end()
```

```
        if result == 'X':
            return (-1, 0, 0)
        elif result == 'O':
            return (1, 0, 0)
        elif result == '.':
            return (0, 0, 0)

        for i in range(0, 3):
            for j in range(0, 3):
                if self.current_state[i][j] == '.':
                    self.current_state[i][j] = 'X'
                    (m, max_i, max_j) = self.max()
                    if m < minv:
                        minv = m
                        qx = i
                        qy = j
                    self.current_state[i][j] = '.'

        return (minv, qx, qy)

    def play(self):
        while True:
            self.draw_board()
            self.result = self.is_end()

            # Printing the appropriate message if the game has ended
            if self.result != None:
                if self.result == 'X':
                    print('The winner is X!')
                elif self.result == 'O':
                    print('The winner is O!')
                elif self.result == '.':
                    print("It's a tie!")

                self.initialize_game()
                return

            # If it's player's turn
            if self.player_turn == 'X':

                while True:

                    start = time.time()
```

```
            (m, qx, qy) = self.min()
            end = time.time()
            print('Evaluation time: {}s'.format(round(end - start, 7)))
            print('Recommended move: X = {}, Y = {}'.format(qx, qy))

            px = int(input('Insert the X coordinate: '))
            py = int(input('Insert the Y coordinate: '))

            (qx, qy) = (px, py)

            if self.is_valid(px, py):
                self.current_state[px][py] = 'X'
                self.player_turn = 'O'
                break
            else:
                print('The move is not valid! Try again.')

        # If it's AI's turn
        else:
            (m, px, py) = self.max()
            self.current_state[px][py] = 'O'
            self.player_turn = 'X'

def main():
    g = Game()
    g.play()

if __name__ == "__main__":
    main()
```

Output:

```
.| .| .|
.| .| .|
.| .| .|

Evaluation time: 5.0726919s
Recommended move: X = 0, Y = 0
Insert the X coordinate: 0
Insert the Y coordinate: 0
X| .| .|
.| .| .|
.| .| .|

X| .| .|
.| O| .|
.| .| .|

Evaluation time: 0.06496s
Recommended move: X = 0, Y = 1
Insert the X coordinate: 0
Insert the Y coordinate: 1
X| X| .|
.| O| .|
.| .| .|

X| X| O|
.| O| .|
.| .| .|
```

```
Evaluation time: 0.0020001s
Recommended move: X = 2, Y = 0
Insert the X coordinate: 2
Insert the Y coordinate: 0
X| X| O|
.| O| .|
X| .| .|

X| X| O|
O| O| .|
X| .| .|

Evaluation time: 0.0s
Recommended move: X = 1, Y = 2
Insert the X coordinate: 1
Insert the Y coordinate: 2
X| X| O|
O| O| X|
X| .| .|

X| X| O|
O| O| X|
X| O| .|

Evaluation time: 0.0s
Recommended move: X = 2, Y = 2
Insert the X coordinate: 2
Insert the Y coordinate: 2
X| X| O|
O| O| X|
X| O| X|

It's a tie!
```

Algorithm: - Alpha Beta Pruning

* It is an improved minimax using a heuristic
* It stops evaluating more when it makes sure that it's worse than previously examined move.
* It maintains two values

Alpha → Best already explored option for player Max

Beta → Best already explored option for player Min.

Initially $\alpha = -\infty$, $\beta = \infty$

* Alpha Beta pruning makes a major difference in evaluating large and complex game tree and it takes more time to recommend the move in first turn.

Alpha-Beta Pruning:

```
import time

class Game:
  def __init__(self):
    self.initialize_game()

  def initialize_game(self):
    self.current_state = [['.','.','.'],
                          ['.','.','.'],
                          ['.','.','.']]
    # Player X always plays first
    self.player_turn = 'X'

  def draw_board(self):
    for i in range(0, 3):
      for j in range(0, 3):
        print('{}|'.format(self.current_state[i][j]), end=" ")
      print()
    print()

  # Determines if the made move is a legal move
  def is_valid(self, px, py):
    if px < 0 or px > 2 or py < 0 or py > 2:
      return False
    elif self.current_state[px][py] != '.':
      return False
    else:
      return True

  # Checks if the game has ended and returns the winner in each case
  def is_end(self):
    # Vertical win
    for i in range(0, 3):
      if (self.current_state[0][i] != '.' and
        self.current_state[0][i] == self.current_state[1][i] and
        self.current_state[1][i] == self.current_state[2][i]):
        return self.current_state[0][i]

    # Horizontal win
    for i in range(0, 3):
```

```
        if (self.current_state[i] == ['X', 'X', 'X']):
            return 'X'
        elif (self.current_state[i] == ['O', 'O', 'O']):
            return 'O'

    # Main diagonal win
    if (self.current_state[0][0] != '.' and
        self.current_state[0][0] == self.current_state[1][1] and
        self.current_state[0][0] == self.current_state[2][2]):
        return self.current_state[0][0]

    # Second diagonal win
    if (self.current_state[0][2] != '.' and
        self.current_state[0][2] == self.current_state[1][1] and
        self.current_state[0][2] == self.current_state[2][0]):
        return self.current_state[0][2]

    # Is whole board full?
    for i in range(0, 3):
        for j in range(0, 3):
            # There's an empty field, we continue the game
            if (self.current_state[i][j] == '.'):
                return None

    # It's a tie!
    return '.'

def max_alpha_beta(self, alpha, beta):
    maxv = -2
    px = None
    py = None

    result = self.is_end()

    if result == 'X':
        return (-1, 0, 0)
    elif result == 'O':
        return (1, 0, 0)
    elif result == '.':
        return (0, 0, 0)

    for i in range(0, 3):
        for j in range(0, 3):
            if self.current_state[i][j] == '.':
```

```
                    self.current_state[i][j] = 'O'
                    (m, min_i, in_j) = self.min_alpha_beta(alpha, beta)
                    if m > maxv:
                        maxv = m
                        px = i
                        py = j
                    self.current_state[i][j] = '.'

                    # Next two ifs in Max and Min are the only difference between regular algorithm and minimax
                    if maxv >= beta:
                        return (maxv, px, py)

                    if maxv > alpha:
                        alpha = maxv

        return (maxv, px, py)


    def min_alpha_beta(self, alpha, beta):

        minv = 2

        qx = None
        qy = None

        result = self.is_end()

        if result == 'X':
            return (-1, 0, 0)
        elif result == 'O':
            return (1, 0, 0)
        elif result == '.':
            return (0, 0, 0)

        for i in range(0, 3):
            for j in range(0, 3):
                if self.current_state[i][j] == '.':
                    self.current_state[i][j] = 'X'
                    (m, max_i, max_j) = self.max_alpha_beta(alpha, beta)
                    if m < minv:
                        minv = m
                        qx = i
                        qy = j
```

```
                self.current_state[i][j] = '.'

                if minv <= alpha:
                    return (minv, qx, qy)

                if minv < beta:
                    beta = minv

        return (minv, qx, qy)

    def play_alpha_beta(self):
     while True:
        self.draw_board()
        self.result = self.is_end()

        if self.result != None:
            if self.result == 'X':
                print('The winner is X!')
            elif self.result == 'O':
                print('The winner is O!')
            elif self.result == '.':
                print("It's a tie!")


            self.initialize_game()
            return

        if self.player_turn == 'X':

            while True:
                start = time.time()
                (m, qx, qy) = self.min_alpha_beta(-2, 2)
                end = time.time()
                print('Evaluation time: {}s'.format(round(end - start, 7)))
                print('Recommended move: X = {}, Y = {}'.format(qx, qy))

                px = int(input('Insert the X coordinate: '))
                py = int(input('Insert the Y coordinate: '))

                qx = px
                qy = py

                if self.is_valid(px, py):
                    self.current_state[px][py] = 'X'
```

```
                    self.player_turn = 'O'
                    break
                else:
                    print('The move is not valid! Try again.')

        else:
            (m, px, py) = self.max_alpha_beta(-2, 2)
            self.current_state[px][py] = 'O'
            self.player_turn = 'X'


def main():
    g = Game()
    g.play_alpha_beta()

if __name__ == "__main__":
    main()
```

Output:

```
In [1]: runfile('D:/18C102/Artificial Intelligence Lab/tic_tac_toe_4.py', wdir='D:/18C102/Artificial Intelligence Lab')
.| .| .|
.| .| .|
.| .| .|

Evaluation time: 0.2000437s
Recommended move: X = 0, Y = 0

Insert the X coordinate: 0

Insert the Y coordinate: 0
X| .| .|
.| .| .|
.| .| .|

X| .| .|
.| O| .|
.| .| .|

Evaluation time: 0.0s
Recommended move: X = 0, Y = 1

Insert the X coordinate: 0

Insert the Y coordinate: 1
X| X| .|
.| O| .|
.| .| .|

X| X| O|
.| O| .|
.| .| .|
```

```
Evaluation time: 0.0s
Recommended move: X = 2, Y = 0

Insert the X coordinate: 2

Insert the Y coordinate: 0
X| X| O|
.| O| .|
X| .| .|

X| X| O|
O| O| .|
X| .| .|

Evaluation time: 0.0s
Recommended move: X = 1, Y = 2

Insert the X coordinate: 1

Insert the Y coordinate: 2
X| X| O|
O| O| X|
X| .| .|

X| X| O|
O| O| X|
X| O| .|
```

```
Evaluation time: 0.0s
Recommended move: X = 2, Y = 2

Insert the X coordinate: 2

Insert the Y coordinate: 2
X| X| O|
O| O| X|
X| O| X|

It's a tie!
```

Result:

Thus, Tic Tac Toe is implemented using minimax algorithm and Alpha beta pruning.